DNS Attacks

ECE/CS 478 Class Project

Grant Lance, Johan Jordaan
Oregon State University, Corvallis, OR, USA, lanceg,author2@oregonstate.edu

I. Introduction

The purpose of this project is to explore the different attacks that leave DNS architecture vulnerable. This project includes a paper and presentation. The slides and any accompanying notes will be included in an appendix. An outline of this paper, we will be covering the basic structure and function of a DNS, then describe the set up and attacks simulated, next talk about possible security measures, and finally share what we have learned from the attacks we studied.

II. DNS - A General Overview

At a very basic level a DNS will take a domain name and return the IP address necessary for communicating with the services registered to this name. Generally, a DNS is a distributed system - there is not a single DNS computer but rather many resolvers that work together. These systems are usually explained using a phone book analogy. The original implementation of DNS can be traced back to ARPANET. During this period hosts were saved on a HOSTS.TXT file at the Network Information Center. It required direct oversight and addresses had to be assigned manually - over the phone. This system evolved into a larger WHOIS directory where the idea of domains were first used. It was during this time that it was chosen to have a domain name reflect the physical location of the computer. In 1983 this system was decentralized to increase the performance needs of the expanding network.

Overall, DNS data storage has the structure of a tree. Each domain name has its own branch. These branches are referred to as delegated subzone. Each branch corresponding to a delegated subzone could contain as many delegated subzones as allowed by the administering name server. The naming convention also ties into the subdomains of a domain name. An example to explain - www.oregonstate.edu is the domain name for OSU. oregonstate is a subdomain of the .edu domain and www is a subdomain of oregonstate. The names themselves are made using a subset of ASCII characters. This set is known by the LDH rule (using only letters, digits and hyphens).

Aside from the DNS server, there may also be a caching resolver that saves the frequent query results of a local user. This information is generally saved by the access point of that individual user. An access point (pineapple) or ISP modem may support recursive queries; for example with www.oregonstate.edu, this means that the resolution would first point to an edu server, then to oregonstate.edu, then to www.oregonstate.edu (Fig 1) [1].

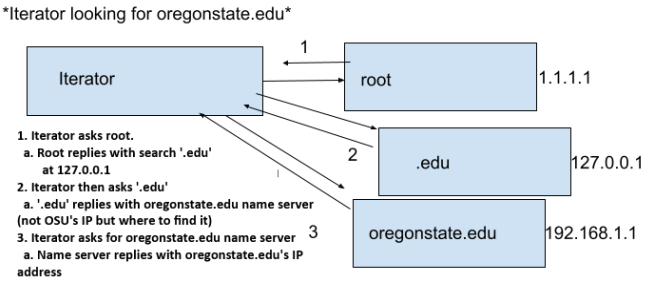

Fig. 1: DNS iterator example

A DNS packet contains a header, a question, and an answer section [2]. The difference between a question and an answer is the overall length; an answer is longer, typically adding a Time-to-Live value and the requested IP address. In the header, there are:

- ID: 16 bits created by the query program, used to match queries to replies.
- QR (Query/Response): 1 bit, 0 for a query, 1 for a response.
- Opcode: 4 bits, specifying the type of query (usually 0 for a standard query).
- AA (Authoritative Answer): 1 bit, used in responses.

The query section contains the domain name being queried, followed by:

- QTYPE: A 16-bit field specifying the type of the query.
- QCLASS: A 16-bit field specifying the class of the query.

For an answer, the packet also includes:

- TTL: Time-to-Live, indicating how long the query should be cached.
- RDATA: The requested IP address.

III. DNS Cache Poisoning Attack

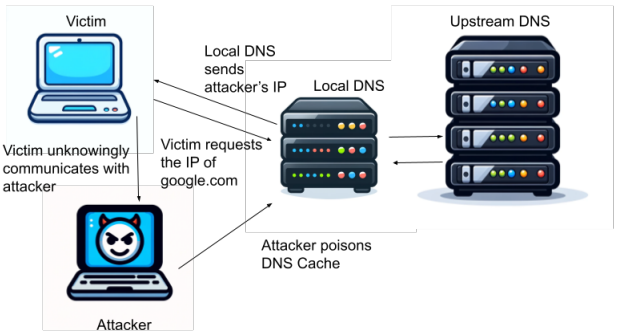

Fig. 2: DNS cache poisoning diagram

In order to simulate DNS cache poisoning, Docker demo from MIT was used [3]. This section will be a technical breakdown of this demo and attack type. Per the license in the readme, redistribution and use of the code are permitted in this manner (academic use). The attack follows this general structure, the user wants to request the IP address of google.com. Without realizing, the IP address for google.com stored by a local DNS server has been corrupted by an attacker. The attacker does this so that the victim will connect to the IP address of their choosing rather than the actual IP address (Fig 2).

Following the guide in the README shows that a fake DNS record can be written into a local DNS cache. As an example, google.com is given a fake/incorrect IP. After running the python script, dig can be called inside the victim container and the cache poisoning can be verified. Any hostname can be spoofed with any desired IP address. To demonstrate, youtube.com was spoofed next with the address 10.0.0.10 (Fig 3).

Using this method the IP address desired by the attacker can be saved into the cache of the victim dns server in the case of google.com the IP address was changed to that of the attacker. In the case of youtube.com, the IP was changed to a different IP address. This was chosen to show that the attacker may not be only interested in directing traffic to their machine but possibly to other machines/servers that are under their control. Using the inspect command on the test network shows that 10.0.0.4 is the IP address for the attacker. Also, the dig output shows that the information has been spoofed. It also confirms that the information came from the local DNS server- showing that the table has been successfully poisoned (Fig 4).

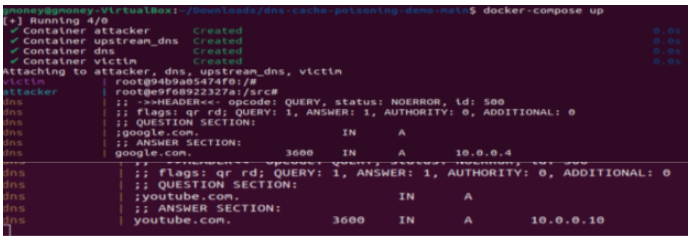

Fig. 3: Script output from poison request

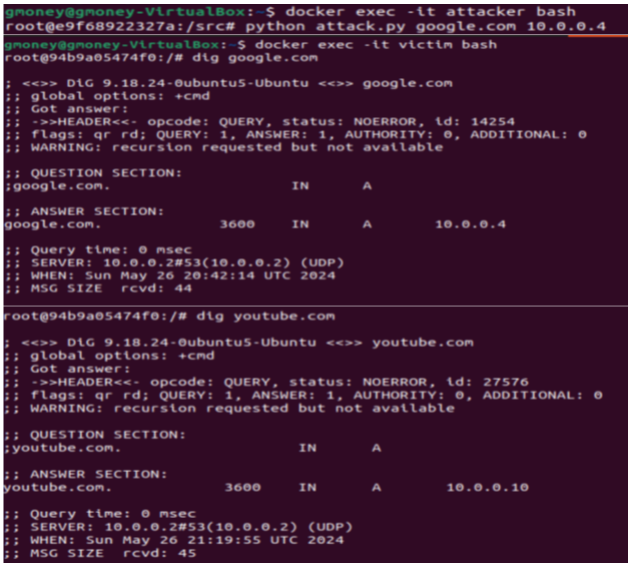

Fig. 4: Dig output showing cache has been poisoned

1) Breakdown of Demo Code

This section of the document will focus on a breakdown of the demo code. Per the licensing this is allowed and will serve as documentation of our learning. The demo uses four containers (Fig 5). Each will be explored from their Dockerfile to function in the network.

Fig. 5: Screenshot of Docker network containers with IPs

- Attacker: the Dockerfile for the attacker container shows that this is a standard docker container with scapy, dnsutils, net-tools, and other packet sniffing stools installed.

The container also contains a python script. This script sends poisoning attacks aimed at the local DNS. This is done by supplying a domain name and fake IP. These are then taken by the container and sent to the local DNS under the guise that they were sent by the Upstream DNS. The script also tries multiple transaction IDs in order to increase the chances of poisoning the cache.

- Local DNS: again standard Dockerfile with the same tools as the attacker. This time instead of the tools being used to launch a poisoning attack on the DNS, the tools are used along with a python script to emulate the function of a local DNS cache. The server will supply an IP that it contains in its cache. If the server does not have a requested IP it will forward the request upstream.
- Victim: a standard container that is used to show the poisoning effects. Contains no scripts or anything special.
- Upstream DNS: Basic model of an upstream DNS server. Has no real functionality except will supply the IP 1.2.3.4 for any DNS query.

Examining the code shows places where security could be introduced to reduce the chances of outside tampering with DNS. Part of this attack relies on the IP of the upstream DNS being known. Having different levels of notoriety among DNS servers is important to maintaining security. Certain servers, like the one upstream, may have a level of control over DNS servers downstream. If that IP is known, an update is easy to forge as demonstrated by this attack demo.

IV. DNS Reply Spoofing

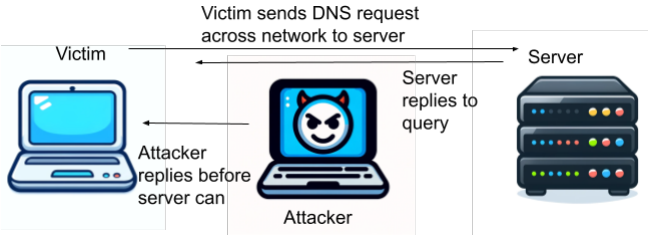

Fig. 6: DNS reply spoofing example

In this scenario the attacker replies to the DNS query sent to the server (Fig 6). Generally, these requests are sent in the clear and other users can see these messages very easily. This is further made easier if the attack occurs on a wireless network. If the victim and server are not using any security it is a very simple task of manufacturing a packet and replying before the server can. As stated previously, DNS packets are sent in the clear, and since the structure of the queries is known; retrieving relevant data from them is somewhat simple [4].

Should an attacker's reply get to a victim before the DNS resolver, the victim may not drop the attacker's response in favor of the DNS resolver. This is dependent on the security between these two parties. However there is generally no security used on DNS queries. This would make it easy for an attacker to field the request and immediately create a response for the victim. In order to carry out this attack, generally only the answer section needs to be modified. In this section the attacker needs to add the desired IP that they wish the victim to connect to. It could be the attacker's machine or one under their control. For the purpose of the demo, the IP address will be that of the attacker.

In order to demonstrate this attack, Docker is used again. This time the code was created by us- no outside demo was used. Three containers are created: a victim, attacker, and server. The network is set up so that the victim uses the server container as its DNS. Usually, port 53 is used for DNS traffic. The attacker will sniff the packets on the network. When the attacker sees a DNS request, it will reply to the sender filling the answer section with its own IP address (Fig 7). This attack relies on the lack of security and the response delay from the DNS.

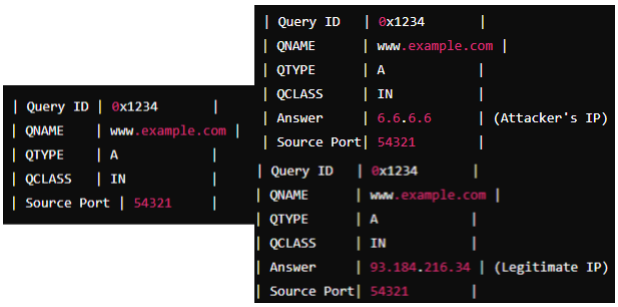

Fig. 7: DNS packet changes from query to reply

Due to the nature of docker containers, we found that simulating the DNS server via a container proved troublesome. This is because this attack largely takes advantage of the delay that a DNS has compared to a local attacker. When both the attacker and DNS are hosted in containers, the delay cannot be taken advantage of because there is no difference in the communication delay between the DNS container and the attacker container. With this now in mind, we opted to display how this could happen between an attacker and victim container. The victim is a user with knowledge of example.com. The IP address of this website is 1.2.3.4. When nslookup example.com is called from the victim container, the attacker container has a script that replies to the DNS request before the server can. This can be seen on the second call of nslookup. The address has been modified by the attacker's response (Fig 9).

The victim refreshes its DNS query generally based on a Time-To-Live (TTL) value. This value is usually around a minute long. In this example here we used nslookup to speed up this process. With the test network connected to the Docker bridge, the container will spoof any DNS traffic it sees. This illustrates the issues we encountered earlier attempting to demo this attack. The attacker is able to spoof the reply if their reply can arrive first. With the container connected to the entire local network of the VM, the container script will block any DNS queries it can (Fig 8). Certain websites do load. However, in our experience this was only Amazon.com and Youtube.com. While resources regarding Amazon's and Google's infrastructure are not publicly available, it can be estimated that there is some level

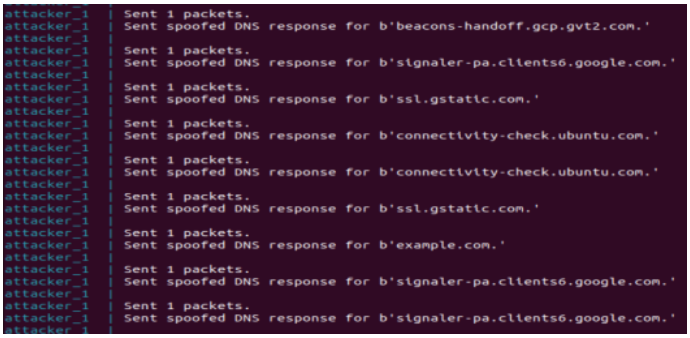

Fig. 8: Attacker script showing attempts to spoof other packets on the network

of security here. Not only because of what we witnessed through our testing, but also because both companies offer their own DNS solutions- Cloud DNS and Route 53. Both are advertised as having security measures [5] [6]. Our tests show the robustness of Amazon's solution. While google.com would load, it was slow and apps were buggy/unresponsive. While using amazon.com there was no almost no noticeable interference and the website functioned normally despite the script attempting to spoof the query. All scripts and dockerfiles for reply spoofing can be found on this github [7].

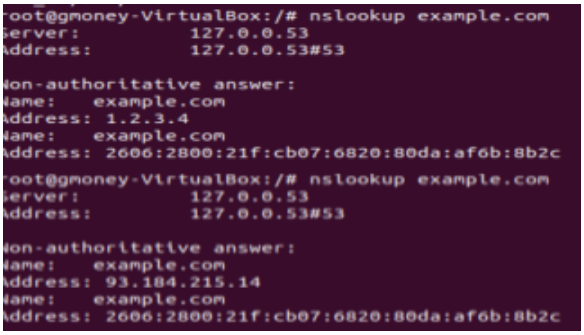

Fig. 9: nslookup showing reply has been spoofed

V. DNS Hijacking

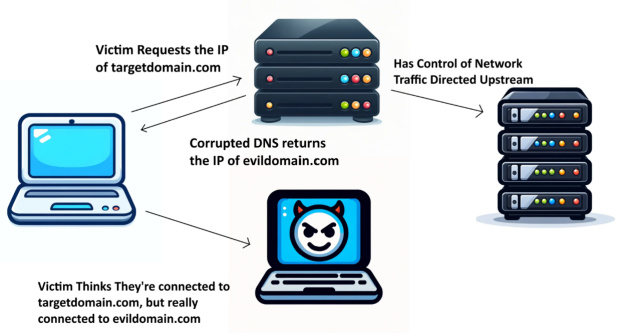

Fig. 10: DNS Hijacking Example

In this scenario the attacker manipulates DNS query responses to redirect the victim's traffic to malicious sites (Fig 10). Hijacking can be accomplished through multiple attack vectors such as intercepting DNS queries, installing malicious software on the victim's computer, or by taking control of the victim's router [8].

For each aforementioned attack vector, the attacker's intent is to direct the victim to a site different from the one the victim has requested. To do so, DNS settings are modified on the compromised device, causing the DNS to return an IP address which directs the victim to an unwanted site. This attack can be used to direct victim's to false web pages representing true websites, leading to users inputting sensitive information such as login credentials or medical information. The website which the victim is directed to could also potentially host its own malware packages which the victim might install unknowingly.

Docker was used to demonstrate how a victim might be redirected unknowingly after a router has already been corrupted by a DNS Hijacking attack. All the code was created specifically for this attack, though it does use standard python and docker libraries. In this attack, the victim will request from the DNS the IP address of 'targetdomain.com', but will instead be redirected to the false web server set up by the attacker. When the corrupted DNS receives the request from the victim for 'targetdomain.com', the DNS will return the IP address 117.18.0.2, which is the IP address of the attacker's malicious web service.

```
#dnsmasq.conf
no-resolv
server=8.8.8.8
address=/targetdomain.com/172.18.0.2
```

Fig. 11: DNS Configuration for Hijacking Example

After the DNS has been setup (Fig 11), the Attacker is begins to execute a custom python script creating a malicious web server at the IP address 117.18.0.2 using the Flask library in python (Fig 12). This simple web server has 2 roles, logging any connections, and sending out the message "This is an attack!". A more malicious web server is likely to pretend to be a login page for the target website, carry malicious software to upload to the victim, or display other malicious content.

```
# attacker.py

from flask import Flask, request
import logging

app = Flask(__name__)

logging.basicConfig(level=logging.INFO)

@app.route('/')
def malicious_content():
    app.logger.info(f"Received request from {request.remote_addr}")
    return "This is an attack!"

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=80)
```

Fig. 12: Attacker Python Script Using Flask

When the victim goes to connect to 'targetdomain.com', they are provided from the DNS an IP address, which they

must assume to be correct. The victim then connects to this IP address (which belongs to the malicious web service run by the attacker), and now receives the message "This is an attack!" (Fig 13). This demonstration shows that the victim was directed by the compromised DNS server to the domain set up by the attacker.

VI. Dynamic DNS Abuse Attack

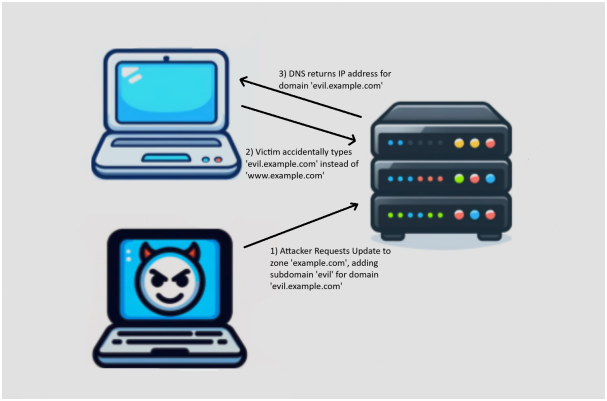

Fig. 13: Dynamic DNS Abuse Attack Example

A Dynamic DNS Abuse Attack takes advantage of an additional feature some DNS services offer called Dynamic Addressing. This feature allows owners of domains to dynamically request an alteration to the existing addresses stored on the DNS [9]. For example, if a web server named example.com wanted to add an inventory page such as example.inventory.com, they might request that the subdomain is added with the IP address 93.120.0.0. A Dynamic DNS Abuse Attack takes advantage of poor protection a DNS might implement against unwanted users changing the DNS records of a protected domain. The attacker finds the vulnerability in the DNS protection, and requests a malicious change to the DNS records. One potential change the attacker might make, for example, is a request to change an existing domain's address to a malicious IP address, or adding a subdomain to catch specific misspellings of the target domain such as gogle.com where the target domain is google.com.

```
options {
    directory "/var/cache/bind";
    recursion yes;
    allow-query { any; };
};

include "/etc/bind/named.conf.local";
```

Fig. 14: DNS configuration file named.conf

To simulate the Dynamic DNS Abuse attack, a very simple DNS was built using the common Docker DNS library called BIND. The DNS was configured in the main configuration file named.conf to allow all queries. This is typical for small or hobby DNS setup where you might want to allow any device to request an IP address of a domain (fig 14). In the local configuration file 'named.conf.local', the zone 'example.com' was set up to allow any requests for a DNS address update (Fig 15).

```
zone "example.com" {
    type master;
    file "/etc/bind/db.example.com";
    allow-update { any; };
};
```

Fig. 15: DNS configuration file named.conf.local

```
#!/bin/bash

# Wait for DNS server to start
sleep 10

# Dynamic DNS update
nsupdate << EOF
server dns-server
zone example.com
update add evil.example.com. 600 IN A 192.168.1.100
send
EOF
```

Fig. 16: Attacker Request Update Adding evil subdomain

Although allowing any device to query an IP address might make sense for some hobby or private use, allowing any request to update the DNS addresses is very dangerous. In this example, the attacker container requests an update to the 'example.com' zone to include the sub-domain 'evil' (fig 16). Although a typo where a victim accidentally types 'evil' instead of 'www' when trying to connect to 'www.example.com' is unlikely, such an error would now result in the victim being sent the IP address of an server controlled by the attacker, as can be seen below (fig 17).

```
jordaanj@pandabox:~$ sudo docker exec -it victim /bin/bash
root@151cbecf85e7:/usr/src/app# nslookup evil.example.com dns-server
Server:         dns-server
Address:        172.19.0.3#53

Name:   evil.example.com
Address: 192.168.1.100

root@151cbecf85e7:/usr/src/app#
```

Fig. 17: Victim Receives evil Subdomain IP Address

Including protection against Dynamic DNS Abuse Attacks is imperative as vulnerabilities allow attackers to request any update to the DNS library, including adding, removing, or changing addresses in the DNS library. Even without a typo, a victim could potentially be directed to a completely malicious website when trying to connect to a domain.

VII. Protecting Against Dynamic Abuse Attacks

Many various defenses exist for protecting against a Dynamic DNS Abuse Attack such as name protection, login credentials, and IP restricted updating. For the sake of this project, it was decided that IP restricted updates would help demonstrate how simple changes to the configuration of the DNS could protect against potential disastrous modifications made by an attacker [10].

Rather than allowing any update requests for the zone 'example.com', a list of trusted IP addresses was created for the DNS to reference when receiving a Dynamic Update Request. This example shows 3 IP addresses were included on the trusted list: 192.168.1.10, 192.168.1.11, and 192.168.1.12 (Fig 18). These IP addresses could, for example, belong to trusted parties in a companies IT department, or for personal use cases, trusted protected devices from which configurations might want to be made.

```
// Define the trusted IP addresses
acl "trusted-updaters" {
    192.168.1.10;  // Example IP address
    192.168.1.11;  // Another example IP address
    192.168.1.12;  // Add as many IP addresses as needed
};

// Define a master zone for example.com
zone "example.com" {
    type master;
    file "/etc/bind/db.example.com";  // Zone file for example.com
    allow-update { trusted-updaters; };  // Only allow updates from trusted IP addresses
};

// Define another master zone for example.net
zone "example.net" {
    type master;
    file "/etc/bind/db.example.net";  // Zone file for example.net
    allow-update { trusted-updaters; };  // Only allow updates from trusted IP addresses
};
```

Fig. 18: Authorization Updates In named.config.local

With this protection added, any devices not belonging to the specific IP addresses or nets will not be able to request a dynamic update for the protected zone on the DNS. As a result, because the attacker does not belong to an approved IP address, when they attempt an update to the DNS, no changes are made to the the DNS library. The victim might again make the mistake of typing the subdomain 'evil.example.com' instead of 'www.example.com', but this time, the DNS will have no history of the subdomain, since it was never added despite the attacker's best efforts (Fig 19).

```
jordaanj@pandabox:~/Docker/Dockerfiles_dynamic_attack$ sudo docker exec -it victim /bin/bash
root@151cbecf85e7:/usr/src/app# nslookup evil.example.com dns-server
Server:         dns-server
Address:        172.19.0.3#53

** server can't find evil.example.com: NXDOMAIN

root@151cbecf85e7:/usr/src/app# nslookup www.example.com dns-server
Server:         dns-server
Address:        172.19.0.3#53

Name:   www.example.com
Address: 127.0.0.1
```

Fig. 19: Victim Protected From evil Subdomain

With an IP based protection scheme on the DNS configuration, it is possible for the attacker to spoof their IP address. For this reason, it makes sense for the DNS to send a confirmation request to the requesting IP address. In such an example, the request for confirmation will go to the IP address from which the request appeared to originate, and not the attacker who spoofed their IP address. Further protections such as authentication protocols like login credentials allow for an even greater degree of security, and is strongly recommended for any DNS implementation.

VIII. Conclusion

A Domain Name System (DNS) translates human-readable domain names, such as google.com, into IP addresses, which are used by routers to connect a user's device to the target domain. Because DNS systems act as the root node through which IP addresses are sourced, ensuring that their data is correct and legitimate is incredibly important. Faulty IP addresses may lead to web servers containing harmful files, fake login pages, or other unwanted content. This document explored potential risk factors associated with DNS, including DNS cache poisoning, DNS hijacking, DNS reply spoofing, and dynamic DNS abuse attacks. Additionally, one potential vector of protection against DNS abuse attacks was explored, where users were authorized based on IP addresses, preventing attackers from modifying the DNS records. Maintaining awareness and exercising safe practices are essential to ensuring a safer web experience.

IX. Individual Contributions

Grant Lance created the material for the following sections in the paper, demo, and presentation

- Introduction
- DNS - A General Overview
- DNS Cache Poisoning Attack
- DNS Reply Spoofing

Johan Jordaan created the material for the following sections in the paper, demo, and presentation

- DNS Hijacking Attack
- Dynamic DNS Abuse Attack
- Protecting Against Dynamic Abuse Attack
- Conclusion

X. References

References

[1] "Domain name system," https://en.wikipedia.org/wiki/Domain_Name_System#:~:text=The%20Domain%20Name%20System%20, accessed: 2024-05-28.

[2] A. Mislove, "Primer on the dns and its security issues," https://mislove.org/teaching/cs4700/spring11/handouts/project1-primer.pdf, 2011, accessed: 2024-05-28.

[3] zphw, "Dns cache poisoning demo," https://github.com/zphw/dns-cache-poisoning-demo, accessed: 2024-05-28.

[4] "Dns spoofing," https://www.proofpoint.com/us/threat-reference/dns-spoofing, accessed: 2024-05-28.

[5] "Amazon route 53," https://aws.amazon.com/route53/, accessed: 2024-05-28.

[6] "Google public dns," https://developers.google.com/speed/public-dns, accessed: 2024-05-28.

[7] "Ece-478 - final project," https://github.com/gmlance/ECE-478---Final-Project/tree/main, accessed: 2024-06-08.

[8] "What is dns hijacking," https://www.fortinet.com/resources/cyberglossary/dns-hijacking#:~:text=Domain%20Name%20Server%20(DNS)%20hijacking,to%20carry%20out%20the%20attack., accessed: 2024-05-28.

[9] "What is dynamic dns," https://www.paloaltonetworks.com/cyberpedia/what-is-dynamic-dns#:~:text=Attackers%20can%20leverage%20DDNS%20services,part%20of%20their%20payload%20distribution., accessed: 2024-05-28.

[10] "Spoofing dns records," https://www.akamai.com/blog/security-research/spoofing-dns-by-abusing-dhcp, accessed: 2024-05-28.